

1

.NET Applications and Tools

WHAT'S IN THIS CHAPTER?

- From .NET Framework to .NET Core to .NET
- .NET **terms**
- .NET **support length**
- **Application** types and technologies
- **Developer** tools
- Using the .NET **command-line interface**
- Programming “**Hello World!**”
- **Technologies** for creating **web apps**

CODE DOWNLOADS FOR THIS CHAPTER

The source code for this chapter is available on the book page at www.wiley.com. Click the Downloads link. The code can also be found at <https://github.com/ProfessionalCSharp/ProfessionalCSharp2021> in the directory `1_CS/HelloWorld`.

The code for this chapter is divided into the following major examples:

- HelloWorld
- WebApp
- SelfContainedHelloWorld

FROM .NET FRAMEWORK TO .NET CORE TO .NET

The first version of .NET was released in **2002**. Since the first version, many things have changed. The first era of .NET was the **.NET Framework** that offered **Windows Forms** for **Windows desktop development**

and **Web Forms** to create **web applications**. This version of .NET was available **only for** Microsoft Windows. At that time, Microsoft also invented a standard for C# at **ECMA** (<https://www.ecma-international.org/publications/standards/Ecma-334.htm>).

Later, **Silverlight** used a subset of this technology with a **limited library** and **runtime running in browsers** using a **browser add-in**. At that time, the company **Ximian** developed the **Mono runtime**. This runtime was available for **Linux** and **Android** and offered a **subset** of Microsoft .NET's **functionality**. Later, **Novell** bought Ximian, and Novell was later bought by **The Attachmate Group**. As the new organization lost interest in .NET, Miguel de Icaza (the founder of Ximian) started **Xamarin** and took the interesting .NET parts into his new organization to start **.NET for Android and iOS**. Nowadays, Xamarin **belongs to Microsoft**, and the Mono runtime is part of the dotnet runtime repo (<https://github.com/dotnet/runtime>).

Silverlight started .NET development for other devices with different form factors, which have different needs for .NET. Silverlight was not successful in the long term because HTML5 offered features that previously only were available by using browser add-ins. However, Silverlight started moving .NET in other directions that resulted in .NET Core.

.NET Core was the biggest change to .NET since its inception. .NET code became open-source, you could create apps for other platforms, and the new code base of .NET is using modern design patterns. The next step is a logical move: the version of .NET after .NET Core 3.1 is **.NET 5**. The *Core* name is removed, and version 4 was skipped to send a message to .NET Framework developers that there's a higher version than .NET Framework 4.8, and it's time to move to .NET 5 for creating new applications.

For developers using .NET Core, the move is an easy one. With existing applications, usually all that needs to be changed is the version number of the target framework. Moving applications from the .NET Framework is not that easy and might require bigger changes. Depending on the application type, more or less change is needed. .NET Core 3.x supports WPF and Windows Forms applications. With these application types, the change can be easy. However, existing .NET Framework WPF applications may have features that cannot be moved that easily to the new .NET. For example, application domains are not supported with .NET Core and .NET 5. Moving Windows Communication Foundation (WCF) services to .NET 5 is not at all easy. The server part of WCF is not supported in the new .NET era. The WCF part of the application needs to be rewritten to ASP.NET Core Web API, gRPC, or another communication technology that fulfills the needs.

With existing applications, it can be useful to stay with the .NET Framework instead of changing to the new .NET because the old framework will still be maintained for many years to come. The .NET Framework is installed with Windows 10, and support for the .NET Framework has a long target that is bound to the support of the Windows 10 versions.

The new .NET and NuGet packages allow Microsoft to provide faster update cycles for delivering new features. It's not easy to decide what technology should be used for creating applications. This chapter helps you with that decision. It gives you information about the different technologies available for creating Windows and web apps and services, offers guidance on what to choose for database access, and helps with moving from old technologies to new ones. You'll also read about the .NET tooling that you can use with the code samples through all the chapters of this book.

.NET TERMS

Before digging deeper, you should understand concepts and some important .NET terms, such as what's in the .NET SDK and what the .NET runtime is. You also should get a better understanding of the .NET Framework and .NET, when to use the .NET Standard, and the NuGet packages and .NET namespaces.

.NET SDK

For **developing .NET applications**, you need to install the .NET SDK. The SDK contains the **.NET command-line interface (CLI)**, **tools, libraries**, and the **runtime**. With the .NET CLI, you can create new applications based on

templates, restore packages, build and test the application, and create deployment packages. Later in this chapter in the section “.NET CLI,” you will see how to create and build applications.

If you use Visual Studio 2019, the .NET SDK is installed as part of Visual Studio. If you don’t have Visual Studio, you can install the SDK from <https://dot.net>. Here, you can find instructions on how to install the SDK on Windows, Mac, and Linux systems.

You can install multiple versions of the .NET SDK in parallel. The command

```
> dotnet --list-sdks
```

shows all the different SDK versions that are installed on the system. By default, the latest version is used.

NOTE *To run the command, you have many different options to start a command prompt. One is the Windows built-in Command Prompt; you can install the new Windows Terminal; if Visual Studio is installed, you can start the Developer Command Prompt; or you can use the bash shell. Read more on the Windows Terminal later in this chapter in the section “Developer Tools.”*

You can create a `global.json` file if you do not want to use the latest version of the SDK. The command

```
> dotnet new globaljson
```

creates the file `global.json` in the current directory. This file contains the version element with the version number currently used. You can change the version number to one of the other SDK versions that is installed:

```
{
  "sdk": {
    "version": "5.0.202"
  }
}
```

In the directory of `global.json` and its subdirectories, the specified SDK version is used. You can verify this with

```
> dotnet --version
```

.NET Runtime

On the target system, the .NET SDK is not required. Here you just need to install the .NET runtime. The runtime includes all the core libraries and the dotnet driver.

The dotnet driver is used to run the application—for example, the Hello, World application with

```
> dotnet hello-world.dll
```

At <https://dot.net>, you can find not only instructions to download and install the SDK on different platforms but also the runtime.

Instead of installing the runtime on the target system, you also can deliver the runtime as part of the application (which is known as *self-contained deployment*). This technique is very different from older .NET Framework applications and is covered later in the chapter in the “Using the .NET CLI” section.

To see which runtimes are installed, you can use

```
> dotnet --list-runtimes
```

Common Language Runtime

The C# compiler **compiles** C# code to **Microsoft Intermediate Language (IL)** code. This code is a little bit like **assembly code**, but it has more **object-oriented features**. The IL code is run by the **Common Language Runtime (CLR)**. What's done by a CLR?

The IL code is compiled to **native code** by the CLR. The IL code available in .NET assemblies is compiled by a **Just-In-Time (JIT) compiler**. This compiler creates **platform-specific native code**. The runtime includes a JIT compiler named **RyuJIT**. This compiler is **not only faster** than the previous one, but it also has **better support** for using Edit & Continue **while you're debugging** the application with Visual Studio.

The runtime also includes a type system with a type loader that is responsible for loading types from assemblies. Security infrastructure with the type system verifies whether certain type system structures are permitted—for example, with inheritance.

After instances of types are created, they also need to be destroyed, and memory needs to be recycled. Another feature of the runtime is the garbage collector. The garbage collector cleans up memory from objects that are no longer referenced in the managed heap.

The runtime is also responsible for threading. When you are creating a managed thread from C#, it is not necessarily a thread from the underlying operating system. Threads are virtualized and managed by the runtime.

NOTE *How you can create and manage threads from C# is covered in Chapter 17, “Parallel Programming.” Chapter 13, “Managed and Unmanaged Memory,” gives information about the garbage collector and how to clean up memory.*

.NET Compiler Platform

The C# compiler that's installed as part of the SDK belongs to the .NET Compiler Platform, which is also known by the code name **Roslyn**. Roslyn allows you to interact with the compilation process, work with syntax trees, and access the semantic model that is defined by language rules. You can use Roslyn to write code analyzers and refactoring features. You also can use Roslyn with a new feature of **C# 9**, code generators, which are discussed in Chapter 12, “Reflection, Metadata, and Source Generators.”

.NET Framework

The **.NET Framework** is the name of the **old .NET**. The last **version** available is .NET Framework **4.8**. It's **not that useful** to create new applications with this framework, but of course you can **maintain existing applications** because this technology will still be supported for many years to come. If existing applications don't get any advantages by moving to new technologies and there's not a lot of maintenance going on, there's no need to switch in the short term.

Depending on the technologies used with existing applications, the switch to .NET can be easy. WPF and Windows Forms have been offered with newer technologies since .NET Core 3. However, WPF and Windows applications could have used features where the application architecture might need a change.

Examples of technologies that are no longer offered with new versions of .NET are ASP.NET Web Forms, Windows Communication Foundation (WCF), and Windows Workflow Foundation (WF). Instead of ASP.NET Web Forms, you can rewrite applications using ASP.NET Blazor. Instead of WCF, you can use ASP.NET Core Web API or gRPC. Instead of WF, moving to Azure Logic Apps might be useful.

.NET Core

.NET Core is the new .NET that is used by all new technologies and is a main focus of this book (with the new name .NET). This framework is **open source**, and you can find it at <http://www.github.com/dotnet>.

The runtime is the **CoreCLR repository**; the **framework** containing collection **classes, file system access, console, XML**, and a **lot more** is in the CoreFX repository.

Unlike the .NET Framework, where the specific version you needed for the application had to be installed on the system, with .NET Core 1.0, the framework, including the runtime, is delivered with the application. Previously, there were times when you might have had problems deploying an ASP.NET web application to a shared server because the provider had older versions of .NET installed; those times are gone. Now you can deliver the runtime with the application, and you are not dependent on the version installed on the server.

.NET Core is designed in a modular approach. The framework is split into a large list of NuGet packages. So that you don't have to deal with all the packages, *metapackages* reference the smaller packages that work together. This even improved with .NET Core 2.0 and ASP.NET Core 2.0. With ASP.NET Core 2.0, you just need to reference `Microsoft.AspNetCore.All` to get all the packages you typically need with ASP.NET Core web applications.

.NET Core can be updated at a fast pace. Even updating the runtime doesn't influence existing applications because the runtime can be installed with the applications. Now Microsoft can improve .NET Core, including the runtime, with faster release cycles.

NOTE *For developing apps using .NET Core, Microsoft created new command-line utilities. These tools are introduced later in this chapter through a “Hello World!” application in the section “Using the .NET CLI.”*

.NET

Starting with .NET 5, .NET Core has a **new name**: .NET. Removing “Core” from the name should tell developers who still use the .NET Framework that there's not a new version of the .NET Framework from now on. The .NET Framework is no longer receiving new features. **For new applications**, you should use .NET.

.NET Standard

.NET Standard is an **important specification** when **creating and using libraries**. .NET Standard offers a **contract** rather than an implementation. With this contract, available APIs are listed. With every new version of .NET Standard, new APIs are added. APIs are never removed. For example, .NET Standard 2.1 lists more APIs than .NET Standard 1.6.

When you're creating a library, you probably want to use as many APIs as possible, so I suggest you choose the most recent .NET Standard version. However, the highest standard version also means the lowest number of platforms that support this standard, so you may need to take that into consideration.

A table at <https://docs.microsoft.com/dotnet/standard/net-standard> gives you the details on what platform supports which version of the standard. For example, .NET Framework 4.6.1 and later support up to .NET Standard 2.0. In addition, .NET Core 3.0 and later (which includes .NET 5 and later) support .NET Standard 2.1. The Universal Windows Platform build 10.0.16299 supports .NET Standard 2.0. Xamarin.Android 10.0 supports .NET Standard 2.1.

As of .NET 5, the .NET Standard becomes irrelevant. If you're creating libraries with .NET 5, you can use libraries from .NET 5, .NET 6, and later applications. Similarly, when you're creating libraries with .NET 7, you can use libraries from applications written with .NET 7 and later.

However, we can't expect that the .NET Framework, Mono, and other older technologies will just fade away, so .NET Standard will still be needed for many years to come. If you need to support older technologies with your libraries, you'll still need .NET Standard.

NOTE Read detailed information about .NET Standard in Chapter 14, “Libraries, Assemblies, Packages, and NuGet.”

NuGet Packages

In the early days, assemblies were reusable units with applications. That use is still possible (and necessary with some assemblies) when you’re adding a reference to an assembly for using the public types and methods from your own code. However, using libraries can mean a lot more than just adding a reference and using it. Using libraries can also mean making some configuration changes or using scripts to take advantage of some features. The target framework determines which binaries you can use. These are reasons to package assemblies within NuGet packages, which are zip files that contain the assembly (or multiple assemblies) as well as configuration information and PowerShell scripts.

Another reason for using NuGet packages is that they can be found easily; they’re available not only from Microsoft but also from third parties. NuGet packages are easily accessible on the NuGet server at <https://www.nuget.org>.

You can add NuGet packages to applications with the .NET CLI:

```
> dotnet add package <package-name>
```

From the references within a Visual Studio project, you can open the NuGet Package Manager (see Figure 1-1). There you can search for packages and add them to the application. This tool enables you to search for packages that are not yet released (including prerelease options) and define the NuGet server that should be searched for packages. One place to search for packages can be your own shared directory where you’ve placed your internal packages that you’ve used.

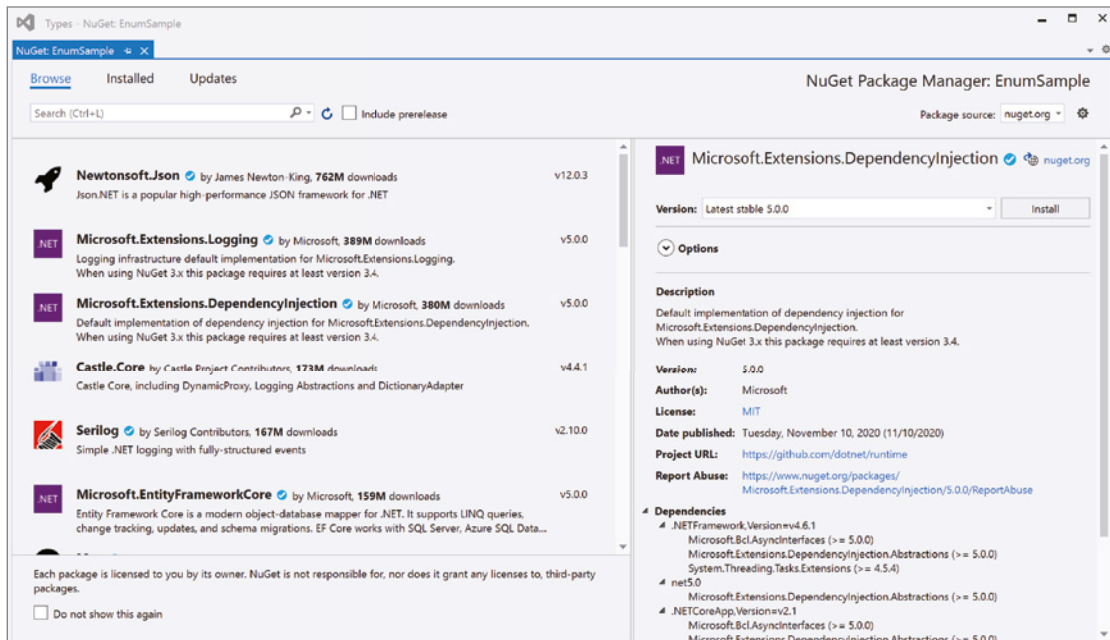


FIGURE 1-1

Namespaces

The **classes** available with .NET are **organized** in **namespaces**. Most of these namespaces start with the name `System` or `Microsoft`. The following table describes a few of the namespaces to give you an idea about the hierarchy:

NAMESPACE	DESCRIPTION
<code>System.Collections</code>	This is the root namespace for collections. Collections are also found within subnamespaces such as <code>System.Collections.Concurrent</code> and <code>System.Collections.Generic</code> .
<code>System.Diagnostics</code>	This is the root namespace for diagnostics information , such as event logging and tracing (in the namespace <code>System.Diagnostics.Tracing</code>).
<code>System.Globalization</code>	This is the namespace that contains classes for globalization and localization of applications.
<code>System.IO</code>	This is the namespace for File input/output (I/O), which includes classes that access files and directories . Readers, writers, and streams are here.
<code>System.Net</code>	This is the namespace for core networking , such as accessing DNS servers and creating sockets with <code>System.Net.Sockets</code> .
<code>System.Threading</code>	This is the root namespace for threads and tasks. Tasks are defined within <code>System.Threading.Tasks</code> .
<code>Microsoft.Data</code>	This is the namespace for accessing databases . <code>Microsoft.Data.SqlClient</code> contains classes that access the SQL Server. The previous classes from <code>System.Data</code> have been repackaged into <code>Microsoft.Data</code> .
<code>Microsoft.Extensions.DependencyInjection</code>	This is the namespace for the Microsoft DI container that is part of .NET.
<code>Microsoft.EntityFrameworkCore</code>	To access relational and NoSQL databases , Entity Framework Core can be used. Types are defined in this namespace.

.NET SUPPORT LENGTH

When you're working in the new era of .NET, you should know about versions with different support cycles. .NET releases differ based on a **Current** or **Long-Term Support** LTS moniker. LTS versions are supported at least three years, or for one year after the next LTS version is available. If for example, the next LTS version is available **2.5 years** after the previous one was released, and the previous one has a support length of **3.5 years**.

Current versions are supported for only three months after the next version is available. At the time of this writing, .NET Core 2.2 and 3.0 are current versions that are already no longer supported with security and hot fixes, whereas .NET Core 2.1 and 3.1 are LTS versions that still have support. The following table lists the .NET Core and .NET versions with their release dates, support level, and end-of-life dates:

.NET CORE/.NET VERSION	RELEASE DATE	SUPPORT LEVEL	END OF LIFE
1.0	June 27, 2016	LTS	June 27, 2019
1.1	Nov. 16, 2016	LTS*	June 27, 2019
2.0	Aug. 14, 2017	Current	Oct. 1, 2018
2.1	May 30, 2018	LTS	Aug. 21, 2021
2.2	Dec. 4, 2018	Current	Dec. 23, 2019
3.0	Sep. 23, 2019	Current	Mar. 3, 2020
3.1	Dec. 3, 2019	LTS	Dec. 3, 2022
5.0	Nov. 10, 2020	Current	around Feb. 2022
6.0	Nov. 2021	LTS	Nov. 2024
7.0	Nov. 2022	Current	Feb. 2024 or earlier in case minor versions are released
8.0	Nov. 2023	LTS	Nov. 2026

Starting with .NET 5, the versions become more predictable. Every year in November, a new major release is available. Every second year, the release is an LTS version.

Depending on the environment you're working in, you might decide to use LTS or Current versions. With current versions, you get new features faster, but you need to upgrade to newer versions more often. While the application is in its **active development stage**, you might decide to use the **current version**. As your application is becoming **more stable**, you can switch to the next **LTS version**.

If you already started development with continuous integration/continuous delivery (CI/CD), it can be an easy task to use only current versions and receive new features faster.

APPLICATION TYPES AND TECHNOLOGIES

You can use C# to create console applications; with most code samples in the first chapters of this book, you'll do that exact thing. For many programs, console applications are not used that often. You can use C# to create applications that use many of the technologies associated with .NET. This section gives you an overview of the different types of applications that you can write in C#.

Data Access

Before taking a look at the application types themselves, let's look at technologies that are used by all application types for access to data.

Files and directories can be accessed by using simple API calls; however, the simple API calls are not flexible enough for some scenarios. With the Stream API, you have a lot of flexibility, and the streams offer many more features, such as encryption or compression. Readers and writers make using streams easier. All of the different options available here are covered in Chapter 18, “Files and Streams.”

To read and write to databases, you can use an abstraction layer, Entity Framework Core (Chapter 21, “Entity Framework Core”). Entity Framework Core offers a mapping of object hierarchies to the relations of a database. EF Core not only offers using different relational databases but also has support for NoSQL databases, such as Azure Cosmos DB.

Windows Apps

For creating Windows apps, you can use the new UI control WinUI 3.0 to create either Universal Windows Platform (UWP) or Windows desktop applications. UWP applications make use of a sandboxed environment where the application needs to request permissions from the user depending on the APIs used. The desktop application version can be compared to a WPF and Windows Forms application where nearly all .NET 5 APIs can be used. WPF and Windows Forms applications can also be updated to use new modern WinUI controls.

Creating WinUI applications with XAML code using the MVVM pattern is covered in Chapter 30, “Patterns with XAML Apps,” and the chapters that follow it.

NOTE *Creating WinUI apps is covered in Chapter 30 with an introduction to XAML, the different XAML controls, and the lifetime of apps. You can create apps with WinUI, WPF, UWP, Xamarin, the Uno Platform, and Maui by using as much common code as possible by supporting the MVVM pattern. This pattern is covered in Chapter 30. To create cool looks and style the app, be sure to read Chapter 31, “Styling Windows Apps.” Finally, Chapter 32, “Advanced Windows Apps,” dives into some advanced features of Windows apps.*

Web Applications

For creating web applications with .NET, several options are available. A technology that implements the Model-View-Controller (MVC) pattern with the application structure is ASP.NET Core MVC. If you have an existing .NET Framework ASP.NET MVC application, the move to ASP.NET Core MVC shouldn’t be too hard.

ASP.NET Core Razor Pages provide an easier option compared to the MVC pattern. Razor Pages can use code-behind or mix the C# code with the HTML page. This solution is easier to start with, and it also can be used with MVC. The dependency injection features of Razor Pages make it easy to create reusable code.

ASP.NET Core Blazor is a new technology that is used to get rid of JavaScript code. With a server-side variant, user interface events are handled on the server. The client and server are continuously connected using SignalR behind the scenes. Another variant of Blazor is using WebAssembly on the client. With this, you can use C#, HTML, and CSS to write code running binary in the client. Because WebAssembly is an HTML 5 standard, Blazor runs in all modern browsers without the need for an add-in.

The original introduction of ASP.NET fundamentally changed the web programming model. ASP.NET Core changed it again. ASP.NET Core allows the use of .NET Core for high performance and scalability, and it runs not only on Windows but also on Linux systems.

With ASP.NET Core, ASP.NET Web Forms is no longer covered. (ASP.NET Web Forms can still be used and is updated with .NET 4.7.)

ASP.NET Core MVC is based on the well-known MVC pattern for easier unit testing. It also allows a clear separation for writing user interface code with HTML, CSS, and JavaScript, and it uses C# on the back end.

NOTE Chapter 24, “ASP.NET Core,” covers the foundation of ASP.NET Core. Chapter 26, “Razor Pages and MVC,” continues describing the foundation and adds using Razor Pages, Razor Views, and the ASP.NET Core MVC framework. Chapter 27, “Blazor,” continues with Razor components and covers development both for Blazor Server and Blazor WebAssembly.

Services

SOAP and WCF fulfilled their duties in the past. Modern apps make use of **Representational State Transfer (REST)** and the **Web API**. Using ASP.NET Core to create a Web API is an option that is a lot easier for communication and fulfills more than 90 percent of requirements by distributed applications. This technology is based on **REST**, which defines guidelines and best practices for **stateless** and **scalable** web services.

The **client** can receive **JSON or XML data**. JSON and XML can also be formatted in a way to make use of the Open Data (OData) specification.

The features of this new API make it easy to consume from web clients using JavaScript, .NET, and other technologies.

Creating a Web API is a good approach for creating microservices. The approach to build microservices defines smaller services that can run and be deployed independently and have their own control of a data store.

To describe the services, a new standard has been developed—the OpenAPI (<https://www.openapis.org>), which has its roots with Swagger (<https://swagger.io/>).

For remote procedure calls (RPC) like communication, you can use gRPC, which offers a binary communication based on HTTP/2 that can be used across different platforms.

NOTE The ASP.NET Core Web API, the OpenAPI, gRPC, and more information on microservices are covered in Chapter 25, “Services.”

SignalR

For **real-time web functionality** and **bidirectional communication** between the **client** and the **server**, **SignalR** is an ASP.NET Core technology. SignalR allows pushing information to connected clients as soon as information is available. SignalR makes use of the **WebSocket technology** to push information.

NOTE The foundation of SignalR connection management, grouping of connections, and streaming are discussed in Chapter 28, “SignalR.”

Microsoft Azure

Nowadays, you can’t ignore the cloud when considering the development picture. Although this book doesn’t include a dedicated chapter on cloud technologies, **Microsoft Azure** is referenced in several chapters in this book.

Microsoft Azure offers software as a service (**SaaS**), infrastructure as a service (**IaaS**), platform as a service (**PaaS**), and functions as a service (**FaaS**), and sometimes offerings are in between these categories. Let’s take a look at some Microsoft Azure offerings.

Software as a Service

SaaS offers **complete software**; you don't have to deal with **management of servers, updates, and so on**. Office 365 is one of the **SaaS offerings** for using email and other services via a **cloud offering**. A SaaS offering that's relevant for developers is **Azure DevOps Services**. **Azure DevOps Services** is the cloud version of **Azure DevOps Server** (previously known as Team Foundation Server) that can be used for private and public **code repository**, for **tracking bugs and work items**, and for **building and testing services**. Another offering from Microsoft in this category is **GitHub**, which is just enhanced to receive many features from Azure DevOps.

Infrastructure as a Service

Another service offering is IaaS. **Virtual machines** are included in this service offering. You are responsible for **managing** the operating system and maintaining updates. When you create virtual machines, you can decide between **different hardware offerings** starting with shared cores up to 416 cores (at the time of this writing, but things change quickly). The M-Series of machines include **416 cores**, 11.4TB RAM, and 8TB local SSD.

With preinstalled operating systems, you can decide between Windows, Windows Server, Linux, and operating systems that come preinstalled with SQL Server, BizTalk Server, SharePoint, Oracle, and many other products.

I use virtual machines often for environments that I need only for several hours a week because the virtual machines are paid on an hourly basis. If you want to try compiling and running .NET Core programs on Linux but don't have a Linux machine, installing such an environment on Microsoft Azure is an easy task.

Platform as a Service

For developers, the most relevant part of Microsoft Azure is **platform as a service (PaaS)**. You can access services for **storing and reading data**, use **computing and networking capabilities** of app services, and integrate **developer services** within the application.

For storing data in the cloud, you can use a **relational data store SQL Database**. SQL Database is nearly the same as the on-premise version of SQL Server. There are also some NoSQL solutions, such as Cosmos DB, with different store options such as JSON data, relationships, or table storage, and Azure Storage that stores blobs (for example, for images or videos).

App Services can be used to **host your web apps and API apps** that you create with ASP.NET Core.

Along with the previously introduced Visual Studio Team Services, another part of the Developer Services in Microsoft Azure is Application Insights. With faster release cycles, it's becoming more and more important to get information about how the user uses the app. What menus are never used because the users probably can't find them? What paths in the app does the user take to accomplish tasks? With Application Insights, you can get good anonymous user information to find out the issues users have with the application, and, with DevOps in place, you can do quick fixes.

You can also use Cognitive Services that offer functionality to process images, use Bing Search APIs, understand what users say with Language services, and more.

Functions as a Service

FaaS, also known with the category name **Azure serverless**, is a new concept for **cloud service**. Of course, behind the scenes there's **always a server**. You just don't pay for reserved CPU and memory because they're handled with **AppServices** that are used from web apps. Instead, **you pay based on consumption**—the number of calls done with some limitations on the memory and time needed for the activity. Azure Functions is one technology that can be deployed using FaaS.

NOTE Chapter 15, “Dependency Injection and Configuration,” not only describes the architecture to define configuration with .NET applications, but it also covers what you need to use this configuration approach to access Microsoft Azure App Configuration and the Azure Key Vault. Chapter 16, “Diagnostics and Metrics,” covers using Azure Monitor, and Chapter 21 shows how to access relational databases both with the on-premises SQL database and with Azure SQL. It also shows how you can use EF Core to access the Azure Cosmos NoSQL database. Chapter 25 uses Azure App Services and Azure Functions for deployment options.

DEVELOPER TOOLS

For development, you need an **SDK to build your applications** and test them, and you need a **code editor**. Some other tools can help, such as a **Linux environment** on your Windows system and an environment to run Docker images. Let’s get into some **practical tools**.

.NET CLI

For development, you need the .NET SDK. If you’re using Visual Studio for development, the .NET SDK is installed with Visual Studio. If you’re using a different environment or you want to install different versions that are not part of the Visual Studio installation, you can get downloads for the SDK from <https://dot.net>. Here you can download and install distributions of the SDK for different platforms.

Part of the SDK is the .NET CLI—the **command-line interface** to develop .NET applications. You can use the .NET CLI to create new applications, compile applications, run unit tests, create NuGet packages, and create the files you need for publishing. Other than that, you can use any editor such as Notepad to write the code. Of course, if you have access to other tools that offer IntelliSense, using them makes it easier to run and debug your applications.

A tour of the .NET CLI is given later in this chapter in the section “Using the .NET CLI.”

Visual Studio Code

Visual Studio Code is a **lightweight editor** available not only on **Windows** but also on **Linux** and **macOS**. The community created a huge number of extensions that make Visual Studio Code the preferred environment for many technologies.

With many chapters of this book, you can use Visual Studio Code as your development editor. What you currently can’t do is create WinUI and Xamarin applications. You can use Visual Studio Code for .NET Core console applications and ASP.NET Core web applications.

You can download Visual Studio Code from <http://code.visualstudio.com>.

Visual Studio Community

This **edition of Visual Studio is a free edition** with features that the Professional edition previously had, but there’s a license restriction for when it can be used. It’s free for open-source projects and training and to academic and small professional teams. Unlike the Express editions of Visual Studio that previously have been the free editions, this product allows using extensions with Visual Studio.

Visual Studio Professional

Visual Studio Professional includes more features than the Community edition, such as the CodeLens and Team Foundation Server for source code management and team collaboration. With this edition, you also get a subscription that includes several server products from Microsoft for development and testing, as well as a free amount that you can use with Microsoft Azure for development and testing.

Visual Studio Enterprise

Unlike the Professional edition, Visual Studio Enterprise contains a lot of tools for testing, such as Live Unit Testing, Microsoft Fakes (unit test isolation), and IntelliTest (unit testing is part of all Visual Studio editions). With Code Clone you can find similar code in your solution. Visual Studio Enterprise also contains architecture and modeling tools to analyze and validate the solution architecture.

NOTE *Be aware that with a Visual Studio subscription you're entitled to free use of Microsoft Azure up to a specific monthly amount that is contingent on the type of the Visual Studio subscription you have.*

NOTE *For some of the features in this book—for example, live unit testing that is briefly explained—you need Visual Studio Enterprise. However, you can work through most parts of the book with the Visual Studio Community edition.*

Visual Studio for Mac

Visual Studio for Mac originated in the Xamarin Studio, but now it has a lot more than the earlier product. The actual version of Visual Studio for Mac is using the same source code for the editor that is available with the Windows version of Visual Studio. With Visual Studio for Mac, you can create not only Xamarin apps but also ASP.NET Core apps that run on Windows, Linux, and Mac. With many chapters of this book, you can use Visual Studio for Mac. Exceptions are the chapters that cover WinUI (Chapters 29 through 31), which require Windows to run and develop the app.

Windows Terminal

After so many years without changes to the Windows command prompt, now there's a completely new one. The source code is public at <https://github.com/Microsoft/terminal>, and it offers many features that are useful for development. This terminal offers multiple tabs and different shells, such as the Windows PowerShell, a command prompt, the Azure Cloud Shell, and WSL 2 environments. You can have the terminal full screen, open different tabs to keep different folders easily accessible, and also split panes to have different folders open in a single screen for easy comparison. New features are added on a monthly basis, and you can install the terminal from the Microsoft Store.

WSL 2

WSL 2 is the second generation of the Windows Subsystem for Linux. With this, the subsystem to run Linux is not only faster, but it also offers practically all Linux APIs.

Using WSL 2, you can install different Linux distributions from the Microsoft Store. If you use the Windows Terminal, different tabs can be opened for every Linux distribution installed.

WSL 2 gives you an easy way to build and run .NET applications on a Linux environment from your Windows system. You can even use Visual Studio to debug your .NET applications while they run in the Linux environment. You just need to install the extension .NET Core Debugging with WSL 2. When you run a debug session from Visual Studio, the .NET SDK gets automatically installed in your WSL 2 environment.

Docker Desktop

The Docker Desktop for Linux (which you can install from <https://hub.docker.com/editions/community/docker-ce-desktop-windows>) allows running Docker containers for Linux or Windows. Using Docker allows creating images that include your application code based on images containing the .NET runtime. The .NET runtime itself is based on Linux or Windows images.

You can use Docker to create a solution using many .NET services running in multiple Docker containers. Docker containers are running instances of Docker images that you can build with support from Visual Studio or dotnet tools such as `tye` (<https://github.com/dotnet/tye>).

NOTE *Creating microservices and running them in Docker containers is covered in Chapter 25.*

USING THE .NET CLI

With many chapters in this book, you don't need Visual Studio. Instead, you can use any editor and a command line, such as the .NET CLI. Let's take a look at how to set up your system and how you can use this tool. This works the same on all platforms.

Nowadays, having a focus on the command line is also due to CI/CD. You can create a pipeline in which compiling, testing, and deployment happens automatically in the background.

If you install .NET CLI tools, you have what you need as an entry point to start all these tools. Use the command

```
> dotnet --help
```

to see all the different options of the dotnet tools available. Many of the options have a shorthand notation. For help, you can also type

```
> dotnet -h
```

Creating the Application

The dotnet tools offer an easy way to create a “Hello World!” application. Just enter this command to create a console application:

```
> dotnet new console --output HelloWorld
```

This command creates a new `HelloWorld` directory and adds the source code file `Program.cs` and the project file `HelloWorld.csproj`. The command `dotnet new` also includes the functionality of `dotnet restore` where all needed NuGet packages are downloaded. To see a list of dependencies and versions of libraries used by the application, you can check the file `project.assets.json` in the `obj` subdirectory. Without using the option `--output` (or `-o` as shorthand), the files would be generated in the current directory.

The generated source code looks like the following code snippet:

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

NOTE Since the 1970s, when Brian Kernighan and Dennis Ritchie wrote the book *The C Programming Language*, it's been a tradition to start learning programming languages using a “Hello World!” application. With the .NET CLI, this program is automatically generated.

Let's get into the syntax of this program. The `Main` method is the entry point for a .NET application. The CLR invokes a static `Main` method on startup. The `Main` method needs to be put into a class. Here, the class is named `Program`, but you could call it by any name.

`Console.WriteLine` invokes the `WriteLine` method of the `Console` class. The `Console` class can be found in the `System` namespace. To avoid writing `System.Console.WriteLine` to invoke this method, the `System` namespace is opened with the `using` declaration on top of the source file.

After writing the source code, you need to compile the code to run it. How you can do this is explained soon in the section “Building the Application.”

The created project configuration file is named `HelloWorld.csproj`. This file contains the project configuration, such as the target framework, and the type of binary to create. An important piece of information in this file is the reference to the SDK (project file `HelloWorld/HelloWorld.csproj`):

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

Top-Level Statements

C# 9 allows you to simplify the code for the “Hello World!” application. With *top-level statements*, the `namespace`, `class`, and `Main` method declarations can be removed to write only top-level statements. The application can look like the “Hello World!” application code shown here (code file `HelloWorld/Program.cs`):

```
using System;

Console.WriteLine("Hello World!");
```

If you prefix the invocation of the `WriteLine` method to add the namespace, you can write the program in a single code line:

```
System.Console.WriteLine("Hello World!");
```

NOTE Behind the scenes, with top-level statements, a class and a Main method are still created. Looking into the generated IL code, a class named `<Program>$`, and a main method named `<Main>$` are generated to contain the top-level statements. You just don't have to write this code on your own.

With small applications like sample applications, top-level statements reduce the required code. When C# is used in a script-like environment, top-level statements are practical as well. Top-level statements are discussed in more detail in Chapter 2, "Core C#."

Selecting the Framework and Language Versions

Instead of building a binary for just one framework version, you can replace the `TargetFramework` element with `TargetFrameworks`, and you can specify multiple frameworks as shown with .NET 5 and .NET Framework 4.8. The `LangVersion` element is added because the sample application uses the C# 9 code (top-level statements). Without using this attribute, the C# version is defined by the framework version. .NET 5 by default is using C# 9, and .NET Framework 4.8 is using C# 7.3

(project file `HelloWorld/HelloWorld.csproj`):

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFrameworks>net5.0;net48</TargetFrameworks>
    <LangVersion>9.0</LangVersion>
  </PropertyGroup>
</Project>
```

The `Sdk` attribute specifies the SDK that is used by the project. Microsoft ships different SDKs: `Microsoft.NET.Sdk` for console applications, `Microsoft.NET.Sdk.Web` for ASP.NET Core web applications, and `Microsoft.NET.Sdk.BlazorWebAssembly` for web applications with Blazor and WebAssembly.

You don't need to add source files to the project. Files with the `.cs` extension in the same directory and subdirectories are automatically added for compilation. Resource files with the `.resx` extension are automatically added for embedding resources. You can change the default behavior and exclude/include files explicitly.

You also don't need to add the .NET Core package. When you specify the target framework `net5.0`, the meta-package `Microsoft.NETCore.App` that references many other packages is automatically included.

Building the Application

To build the application, you need to change the current directory to the directory of the application and start `dotnet build`. You can see output like the following, which is compiled for .NET 5.0 and .NET Framework 4.8:

```
> dotnet build
Microsoft (R) Build Engine version 16.8.0 for .NET Copyright (C)
Microsoft Corporation. All rights reserved.

Determining projects to restore...
Restored C:\procsharp\Intro\HelloWorld\HelloWorld.csproj (in 308 ms).
HelloWorld -> C:\procsharp\Intro\HelloWorld\bin\Debug\net48\HelloWorld.exe
HelloWorld -> C:\procsharp\Intro\HelloWorld\bin\Debug\net5.0\HelloWorld.dll
```



```
Build succeeded.  
    0 Warning(s)  
    0 Error(s)  
  
Time Elapsed 00:00:02.82
```

NOTE *The commands `dotnet new` and `dotnet build` automatically restore NuGet packages, so you cannot forget to do it. Restoring NuGet packages retrieves libraries that are referenced in the project file from the NuGet server or other servers that are configured in your environment. You can also explicitly restore NuGet packages with `dotnet restore`.*

As a result of the compilation process, you find the assembly containing the IL code of the `Program` class within the `bin/debug/[net5.0|net48]` folders. If you compare the build of .NET Core with .NET 4.8, you will find a DLL containing the IL code with .NET Core and an EXE containing the IL code with .NET 4.8. The assembly generated for .NET Core has a dependency on the `System.Console` assembly, whereas the .NET 4.8 assembly includes the `Console` class in the `mscorlib` assembly.

To build release code, you need to specify the option `--configuration Release` (shorthand `-c Release`):

```
> dotnet build --configuration Release
```

NOTE *The debug build includes debug symbols and the generated code is not optimized for easier debugging. With the release build, the code is optimized for production and runs faster. From time to time during the development phase (before delivering the application for production), you should try the release build because there can be different behaviors not found with debug builds.*

Running the Application

To run the application, you can use the following `dotnet` command:

```
> dotnet run
```

If the project file targets multiple frameworks, you need to tell the `dotnet run` command which framework to use to run the app by adding the option `--framework`. This framework must be configured with the `csproj` file. With the sample application, you should get the following output of the application after the restore information:

```
> dotnet run --framework net5.0  
Hello World!
```

On a production system, you don't use `dotnet run` to run the application; instead, you just use `dotnet` with the name of the library:

```
> dotnet bin/debug/net5.0/HelloWorld.dll
```

The compiler also creates an executable, which does nothing more than load and start the library. You can start the executable as well. How executables are built for publishing is shown in the next steps.

NOTE As you've seen from building and running the "Hello World!" app on Windows, the `dotnet` tools work the same on Linux and macOS. You can use the same .NET CLI commands on either platform.

Creating a Web Application

Similarly to creating a console application, you can also use the .NET CLI to create a web application. As you enter `dotnet new`, you can see a list of templates available.

The command

```
> dotnet new webapp -o WebApp
```

creates a new ASP.NET Core web application with Razor Pages.

The created project file now contains a reference to the `Microsoft.NET.Sdk.Web` SDK. This SDK contains tools and extensions for the project file that are needed to create web applications and services:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

Now using

```
> dotnet build
> dotnet run
```

starts the Kestrel server of ASP.NET Core to listen on port 5000 and 5001. You can open a browser to access the pages returned from this server, as shown in Figure 1-2.

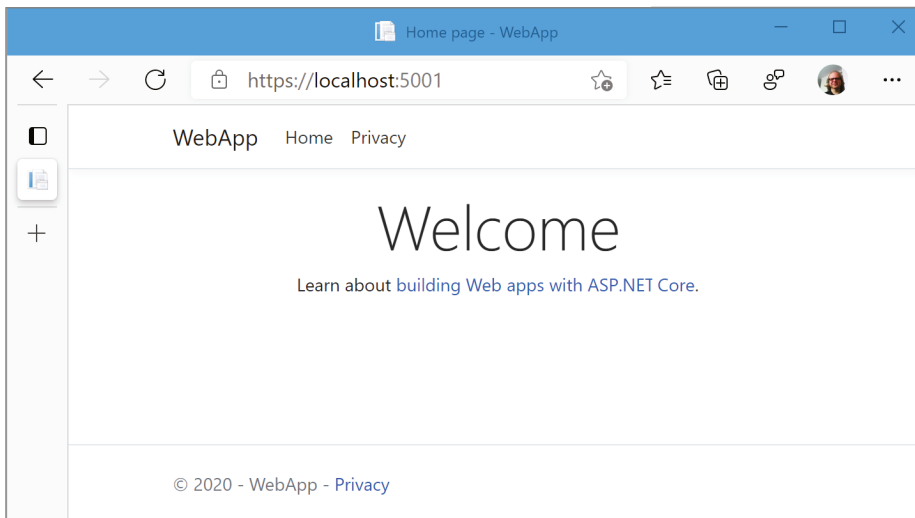


FIGURE 1-2

If you start this for the first time, you're giving a security warning to trust the developer certificate. As you trust the certificate, the warnings will no longer occur.

To stop the application, just press Ctrl+C to send the cancel command.

Publishing the Application

With the `dotnet` tool, you also can create a NuGet package and publish the application for deployment. Let's first create a framework-dependent deployment of the application. This reduces the number of files you need for publishing.

Using the previously created console application, you just need the following command to create the files for publishing. The framework is selected by using `-f`, and the release configuration by using `-c`:

```
> dotnet publish -f net5.0 -c Release
```

You put the files needed for publishing into the `bin/Release/net5.0/publish` directory.

When you use these files for publishing on the target system, you need the runtime as well. You can find the runtime downloads and installation instructions at <https://www.microsoft.com/net/download/>.

NOTE *If your application uses additional NuGet packages, they need to be referenced in the `csproj` file, and the libraries need to be delivered with the application. Read Chapter 14 for more information.*

Self-Contained Deployments

Instead of needing to have the runtime installed on the target system, the `application` can deliver the `runtime` with it. This is known as *self-contained deployment*.

Depending on the platform where the application should be installed, the runtime differs. Thus, with self-contained deployment, you need to specify the platforms supported by specifying `RuntimeIdentifiers` in the project file as shown in the following project file. Here, the runtime identifiers for Windows 10, macOS, and Ubuntu Linux are specified (project file `SelfContainedHelloWorld/SelfContainedHelloWorld.csproj`):

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <PropertyGroup>
    <RuntimeIdentifiers>
      win10-x64;ubuntu-x64;osx.10.11-x64;
    </RuntimeIdentifiers>
  </PropertyGroup>
</Project>
```

NOTE *Get all the runtime identifiers for different platforms and versions from the .NET Core Runtime Identifier (RID) catalog at <https://docs.microsoft.com/en-us/dotnet/core/rid-catalog>.*

Now you can create publish files for all the different platforms:

```
> dotnet publish -c Release -r win10-x64
> dotnet publish -c Release -r osx.10.11-x64
> dotnet publish -c Release -r ubuntu-x64
```

After running these commands, you can find the files needed for publishing in the `Release/[win10-x64|osx.10.11-x64|ubuntu-x64]/publish` directories. As .NET 5.0 runtime is now included, the size of the files needed for publishing has grown. In these directories, you can also find platform-specific executables that can be started directly without using the .NET CLI command.

NOTE *If you're running a Windows System with WSL 2 installed, you can run the binary image created for Ubuntu directly in this subsystem. If you install the .NET SDK within WSL, you can also start the build and publish commands from within the subsystem.*

Creating a Single Executable

Instead of publishing a large list of files, you can create a single executable. Adding the option `-p:PublishSingleFile=true` adds the complete runtime to one binary, which then can be used for deployment. With the following command, a single file is created to the output directory `singlefile`. This directory also contains a file with the `pdb` extension. This file can be deployed to get symbol information for analysis in case the application crashes.

```
> dotnet publish -r win10-x64 -p:PublishSingleFile=true --self-contained
-o singlefile
```

ReadyToRun

To speed up the startup performance of the application, some parts of the application can be precompiled to native code. This way, the IL compiler can reduce its work when running the application. This option can be used with or without `PublishSingleFile`.

```
> dotnet publish -r win10-x64 -p:PublishReadyToRun=true --self-contained
-o readytorun
```

Instead of passing this configuration with the command line, the `<PublishReadyToRun>` element can also be specified in the project file.

Trimming

Of course, a single executable for publishing that includes the complete runtime is large. However, there's a way around that. You can trim all the classes and methods that are not needed for the application to make the binary smaller.

You can specify trimming with the `PublishTrimmed` element in the project file. The `TrimMode` specifies how aggressively trimming should be performed. The value `link` (used in this example) is used to trim based on members and to remove members that are not used. When you set the value to `copyused`, complete assemblies are kept if any of their members are used by the application:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <RuntimeIdentifiers>
```

```
win10-x64;ubuntu-x64;osx.10.11-x64;  
</RuntimeIdentifiers>  
<PublishTrimmed>true</PublishTrimmed>  
<TrimMode>link</TrimMode>  
</PropertyGroup>  
</Project>
```

You use the following command and the previous project configuration to create a single file executable that is trimmed. At the time of this writing, the size of the binary for “Hello, World!” is reduced from 54MB to 2.8MB. That’s quite impressive. As the feature is improved continuously, more savings can be expected in the future.

```
> dotnet publish -o publishtrimmed -p:PublishSingleFile=true --self-contained  
-r win10-x64
```

There is a risk with trimming. For example, if the application makes use of reflection, the trimmer is not aware that the reflected members are needed during runtime. To deal with such issues, you can specify what assemblies, types, and type members should not be trimmed. To configure such options, read the detailed documentation at <https://docs.microsoft.com/dotnet/core/deploying/trimming-options>.

SUMMARY

This chapter covered a lot of ground to review important technologies and changes with .NET. With new applications you should use .NET Core (now renamed to just .NET) for future development. With existing applications, it depends on the state of the application if you prefer to stay with older technologies or migrate to new ones. For moving to .NET, you now know about frameworks that you can use to replace older frameworks.

You read about tools you can use for development and dived into the .NET CLI to create, build, and publish applications.

You looked at technologies for accessing the database and creating Windows apps, and you read about different ways to create web applications.

Whereas this chapter laid the foundation with a “Hello World!” example, Chapter 2 dives fast into the syntax of C#. It covers variables, how to implement program flows, how to organize your code into namespaces, and more.